2
BACKGROUND AND STATE OF THE ART

This chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 we discuss WebAssembly, focusing on its design and security model. Besides, we discuss the current state-of-the-art in the area of WebAssembly program analysis. In Section 2.2 we discuss related works in the area of Software Diversification. Moreover, we delve into the open challenges regarding the diversification of WebAssembly programs.

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the fourth scripting language supported in all major web browser vendors. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?**] in 2017 and extended by Rossberg et al. in 2018 [**?**]. It is designed to be fast, portable, self-contained, and secure.

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup time and memory usage than containerization and virtualization [**?** **?**]. Consequently, in 2019, the Bytecode Alliance proposed WebAssembly System Interface (WASI) [**?**]. WASI pioneered the execution of WebAssembly with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of WebAssembly in heterogeneous platforms [**?**], i.e., IoT and Edge computing [**?** **?**].

Currently, WebAssembly serves a variety of functions in browsers, ranging from gaming to cryptomining [**?**]. Other applications include text processing,

[0] Compilation probe time 2023/10/24 08:05:41

visualization, media processing, programming language testing, online gambling, bar code and QR code fast reading, hashing, and PDF viewing. On the backend, WebAssembly notably excels in short-running tasks. As such, it is particularly suitable for Function as a Service (FaaS) platforms like Cloudflare and Fastly. The subsequent text in this chapter focuses specifically on WebAssembly version 1.0. However, the tools, techniques, and methodologies discussed are applicable to future WebAssembly versions.

2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting WebAssembly program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

```
1  // Static raw data
2  const int A[250];
3
4  // External function usage
5  int ftoi(float a);
6
7  int main() {
8      for(int i = 0; i < 250; i++) {
9          if (A[i] > 100)
10             return A[i] + ftoi(12.54);
11     }
12
13     return A[0];
14 }
```

Listing 2.1: *Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.*

In Listing 2.1 and Listing 2.2, we present a C program alongside its corresponding WebAssembly binary. The C function encompasses various elements such as static allocation, external function usage, and a function definition that includes a loop, conditional branching, function calls, and memory accesses. The Wasm code shown in Listing 2.2 is displayed in its textual format, known as WAT[1]. The static memory declared in line 2 of Listing 2.1 is allocated within the Wasm binary's linear memory, as illustrated in line 47 of Listing 2.2. The function prototype in line 5 of Listing 2.1 is converted into an imported function, as seen in line 8 of Listing 2.2. The main function, spanning lines 7

[1]The WAT text format is primarily designed for human readability and for low-level manual editing.

to 14 in Listing 2.1, is transcribed into a Wasm function covering lines 12 to 38 in Listing 2.2. Within this function, the translation of various C language constructs into Wasm can be observed. For instance, the `for` loop found in line 8 of Listing 2.1 is mapped to a block structure in lines 17 to 31 of Listing 2.2. The loop's breaking condition is converted into a conditional branch, as shown in line 25 of Listing 2.2.

There exist several compilers that turn source code into WebAssembly binaries. For example, LLVM compiles to WebAssembly as a backend option since its 7.1.0 release in early 2019[2], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[3]. Significantly, a study by Hilbig [**?**] reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. The main advantage of using LLVM is that it provides a modular and state-of-the-art optimization infrastructure for WebAssembly binaries. Recently, the Kotlin Multiplatform framework [4] has incorporated WebAssembly as a compilation target, enabling the compilation of Kotlin code to WebAssembly.

A recent trend in the WebAssembly ecosystem involves porting various programming languages by converting both the language's engine or interpreter and the source code into a WebAssembly program. For example, Javy[5] encapsulates JavaScript code within the QuickJS interpreter, demonstrating that direct source code conversion to WebAssembly isn't always required. If an interpreter for a specific language can be compiled to WebAssembly, it allows for the bundling of both the interpreter and the language into a single, isolated WebAssembly binary. Similarly, Blazor[6] facilitates the execution of .NET Common Intermediate Language (CIL) in WebAssembly binaries for browser-based applications. However, packaging the interpreter and the code in one single standalone WebAssembly binary is still immature and faces challenges. For example, the absence of JIT compilation for the "interpreted" code makes it less suitable for long-running tasks [**?**]. On the other hand, it proves effective for short-running tasks, particularly those executed in Edge-Cloud computing platforms.

[2] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`
[3] A subset of the TypeScript language
[4] `https://kotlinlang.org/docs/wasm-overview.html`
[5] `https://github.com/bytecodealliance/javy`
[6] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

```
1  ; WebAssembly magic bytes(\0asm) and version (1.0) ;
2  (module
3  ; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
4    (type (;0;) (func (param f32) (result i32)))
5    (type (;1;) (func))
6    (type (;2;) (func (result i32)))
7  ; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
8    (import "env" "ftoi" (func $ftoi (type 0)))
9  ; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
10   (@custom "name" "...")
11 ; Code section: 0x03 0x00 0x00 0x00 0x5B... ;
12   (func $main (type 2) (result i32)
13     (local i32 i32)
14     i32.const -1000
15     local.set 0
16     block  ;label = @1;
17       loop  ;label = @2;
18         i32.const 0
19         local.get 0
20         i32.add
21         i32.load
22         local.tee 1
23         i32.const 101
24         i32.ge_s
25         br_if 1 ;@1;
26         local.get 0
27         i32.const 4
28         i32.add
29         local.tee 0
30         br_if 0 ;@2;
31       end
32       i32.const 0
33       return
34     end
35     f32.const 0x1.9147aep+3
36     call $ftoi
37     local.get 1
38     i32.add)
39 ; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
40   (memory (;0;) 1)
41 ; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
42   (global (;4;) i32 (i32.const 1000))
43 ; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
44   (export "memory" (memory 0))
45   (export "A" (global 2))
46 ; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
47   (data $data (0) "\00\00\00\00...")
48 ; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
49   (@custom "producers" "...")
50 )
```

Listing 2.2: *Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.*

2.1.2 Extending WebAssembly

The broad spectrum of applicability and the rapid adoption of WebAssembly has resulted in demands for additional features. However, not all of these demands align with its original specifications. Thus, since the introduction of

WebAssembly, various extensions have been proposed for standardization. For instance, the SIMD proposal enables the execution of vectorized instructions in WebAssembly. To become a standard, a proposal must fulfill certain criteria, including having a formal specification and at least two independent implementations, e.g., two different engines. Notably, even after adoption, new extensions are optional, e.g., the core WebAssembly remains untouched and continues to be referred to as version 1.0.

2.1.3 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A WebAssembly binary contains sections of 13 types, each with a specific semantic role and placement within the module. For instance, the *Custom Section* stores metadata like the compiler used to generate the binary, while the *Type Section* contains function signatures that serve to validate the *Function Section*. The *Import Section* lists elements imported from the host, and the *Function Section* details the functions defined within the binary. Other sections like *Table, Memory, and Global Sections* specify the structure for indirect calls, unmanaged linear memories, and global variables, respectively. *Export, Start, Element, Code, Data, and Data Count Sections* handle aspects ranging from declaring elements for host engine access to initializing program state, declaring bytecode instructions per function, and initializing linear memory. Each of these sections must occur only once in a binary and can be empty. For clarity, we also annotate sections as comments in the Wasm code in Listing 2.2.

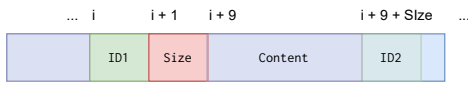


Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

A WebAssembly binary can be processed efficiently due to its organization into a contiguous array of sections. For instance, this structure permits compilers to expedite the compilation process either through parallel parsing or by disregarding *Custom Sections*. Moreover, the *Code Section*'s instructions are further compacted through the use of the LEB128[7] encoding. Consequently, Wasm binaries are not only fast to validate and compile, but also swift to transmit over a network.

[7] `https://en.wikipedia.org/wiki/LEB128`

2.1.4 WebAssembly's runtime

The WebAssembly's runtime characterizes the behavior of WebAssembly programs during execution. This section describes the main components of the WebAssembly runtime, namely the execution stack, functions, memory model, and execution process. These components are crucial for understanding both the WebAssembly's control-flow and the analysis of WebAssembly binaries.

Execution Stack: At runtime, WebAssembly engines instantiate a WebAssembly module. This module is a runtime representation of a loaded and initialized WebAssembly binary described in Section 2.1.3. The primary component of a module instance is its Execution Stack. The Execution Stack stores typed values, labels, and control frames. Labels manage block instruction starts and loop starts. Control frames manage function calls and function returns. Values within the stack can only be static types. These types include `i32` for 32-bit signed integers, `i64` for 64-bit signed integers, `f32` for 32-bit floats, and `f64` for 64-bit floats. Abstract types such as classes, objects, and arrays are not supported natively. Instead, these types are abstracted into primitive types during compilation and stored in linear memory.

Functions: At runtime, WebAssembly functions are closures over the module instance, grouping locals and function bodies. Locals are typed variables that are local to a specific function invocation. A function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the execution stack, writes to the execution stack, or modifies the control-flow of the function. Recalling the example WebAssembly binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 12 and Line 38 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. Notice that, numeric instructions are annotated with its corresponding type. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` in line 38. As the listing also shows, instructions are annotated with a numeric type.

Memory model: A WebAssembly module instance incorporates three key types of memory-related components: linear memory, local variables and global variables. These components can either be managed solely by the host engine or shared with the WebAssembly binary itself. This division of responsibility is often categorized as *managed* and *unmanaged* memory [**?**]. Managed refers to components that are exclusively modified by the host engine at the lowest level, e.g. when the WebAssembly binary is JITed, while unmanaged components can also be altered through WebAssembly opcodes. First, modules may include multiple linear memory instances, which are contiguous arrays of bytes. These are accessed using 32-bit integers (`i32`) and are shareable only between the initiating engine and the WebAssembly binary. Generally, these linear memories

are considered to be unmanaged, e.g., line 21 of Listing 2.2 shows an explicit memory access opcode. Second, there are global instances, which are variables accompanied by values and mutability flags (see example in line 42 of Listing 2.2). These globals are managed by the host engine, which controls their allocation and memory placement completely oblivious to the WebAssembly binary scope. They can only be accessed via their declaration index, prohibiting dynamic addressing. Third, local variables are mutable and specific to a given function instance. They are accessible only through their index relative to the executing function and are part of the data managed by the host engine.

WebAssembly module execution: While a WebAssembly binary could be interpreted, the most practical approach is to JIT compile it into machine code. The main reason is that WebAssembly is optimized and closely aligned with machine code, leading to swift JIT compilation for execution. Browser engines such as V8[8] and SpiderMonkey[9] utilize this strategy when executing WebAssembly binaries in browser clients. Once JITed, the WebAssembly binary operates within a sandboxed environment, accessing the host environment exclusively through imported functions. The communication between the host and the WebAssembly module execution is typically facilitated by trampolines in the JITed machine code.

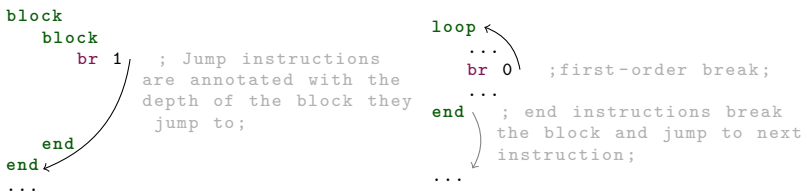
WebAssembly standalone engines: While initially intended for browsers, WebAssembly has undergone significant evolution, primarily due to WASI[**?**]. WASI establishes a standardized POSIX-like interface for interactions between WebAssembly modules and host environments. Compilers can generate WebAssembly binaries that implement WASI, which allows execution in standalone engines. These binaries can then be executed by standalone engines across a variety of environments, including the cloud, servers, and IoT devices [**?**]. Similarly to browsers, these engines often translate WebAssembly into machine code via JIT compilation, ensuring a sandboxed execution process. Standalone engines such as WASM3[10], Wasmer[11], Wasmtime[12], WAVM[13], and Sledge[**?**] have been developed to support both WebAssembly and WASI. In a related development, Singh et al.[**?**] have created a WebAssembly virtual machine specifically designed for Arduino-based devices.

[8] `https://chromium.googlesource.com/v8/v8.git`
[9] `https://spidermonkey.dev/`
[10] `https://github.com/wasm3/wasm3`
[11] `https://wasmer.io/`
[12] `https://github.com/bytecodealliance/wasmtime`
[13] `https://github.com/WAVM/WAVM`

2.1.5 WebAssembly's control-flow

A WebAssembly function groups instructions into blocks, with the function's entrypoint acting as the root block. In contrast to conventional assembly code, control-flow structures in Wasm leap between block boundaries rather than arbitrary positions within the code, effectively prohibiting `goto`s to random code positions. Each block may specify the needed execution stack state before execution as well as the resultant execution stack state once its instructions have been executed. Typically, the execution stack state is simply the quantity and numeric type of values on the stack. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before the execution of the block's instructions. Blocks in Wasm are explicit (see instructions `block` and `end` in lines 16 and 34 of Listing 2.2), delineating where they commence and conclude. By design, a block cannot reference or execute code from external blocks.

During runtime, WebAssembly break instructions can only jump to one of its enclosing blocks. Breaks, except for those within loop constructions, jump to the block's end and continue to the next immediate instruction. For instance, after line 34 of Listing 2.2, the execution would proceed to line 35. Within a loop, the end of a block results in a jump to the block's beginning, thus restarting the loop. For example, if line 30 of Listing 2.2 evaluates as false, the next instruction to be executed in the loop would be line 18. Listing 2.3 provides an example for better understanding, comparing a standard block and a loop block in a Wasm function.



Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. This design hardens the rewriting of WebAssembly binaries. For instance, if an outer block is removed, the depth of the break instructions within nested blocks must be updated to reflect the new enclosing block depth. This is a significant challenge for rewriting tools, as it requires the analysis of the control-flow graph to determine the enclosing block depth for each break instruction.

2.1.6 Security and Reliability for WebAssembly

The WebAssembly ecosystem's expansion needs robust tools to ensure its security and reliability. Numerous tools, employing various strategies to detect vulnerabilities in WebAssembly programs, have been created to meet this need. This paper presents a review of the most relevant tools in this field, focusing on those capable of providing security guarantees for WebAssembly binaries.

Static analysis: SecWasm[**?**] uses information control-flow strategies to identify vulnerabilities in WebAssembly binaries. Conversely, Wasmati[**?**] employs code property graphs for this purpose. Wasp[**?**] leverages concolic execution to identify potential vulnerabilities in WebAssembly binaries. VeriWasm[**?**], an offline verifier designed specifically for native x86-64 binaries JITed from WebAssembly, adopts a unique approach. While these tools emphasize specific strategies, others adopt a more holistic approach. CT-Wasm[**?**], verifies the implementation of cryptographic algorithms in WebAssembly. Similarly, Vivienne applies relational Symbolic Execution (SE) to WebAssembly binaries in order to reveal vulnerabilities in cryptographic implementations[**?**]. For example, both Wassail[**?**] and WasmA[**?**] provide a comprehensive static analysis framework for WebAssembly binaries. However, static analysis tools may have limitations. For instance, a newly, semantically equivalent WebAssembly binary may be generated from the same source code bypassing or breaking the static analysis [**?**]. If the WebAssembly input differs from the input used during sound analysis [**?**], the vulnerability may go unnoticed. Thus, there may be a lack of subjects to evaluate the effectiveness of these tools.

Dynamic analysis: Dynamic analysis involves tools such as TaintAssembly[**?**], which conducts taint analysis on WebAssembly binaries. Fuzzm[**?**] identifies vulnerabilities in host engines by conducting property fuzzing through WebAssembly binary execution. Furthermore, Stiévenart and colleagues have developed a dynamic approach to slicing WebAssembly programs based on Observational-Based Slicing (ORBS)[**?** **?**]. This technique aids in debugging, understanding programs, and conducting security analysis. However, Wasabi[**?**] remains the only general-purpose dynamic analysis tool for WebAssembly binaries, primarily used for profiling, instrumenting, and debugging WebAssembly code. Similar to static analysis, these tools typically analyze software behavior during execution, making them inherently reactive. In other words, they can only identify vulnerabilities or performance issues while pseudo-executing input WebAssembly programs. Thus, facing an important limitation on overhead for real-world scenarios.

Protecting WebAssembly binaries and runtimes: The techniques discussed previously are primarily focused on reactive analysis of WebAssembly binaries. However, there exist approaches to harden WebAssembly binaries, enhancing their secure execution, and fortifying the security of the entire

execution runtimes ecosystem. For instance, Swivel[**?**] proposes a compiler-based strategy designed to eliminate speculative attacks on WebAssembly binaries, particularly in Function-as-a-Service (FaaS) platforms such as Fastly. Similarly, Kolosick and colleagues [**?**] modify the Lucet compiler to use zero-cost transitions, eliminating thee performance overhead of SFI guarantees implementation. Conversely, WaVe[**?**] introduces a mechanized engine for WebAssembly that facilitates differential testing. WaVe can be employed to detect anomalies in engines running Wasm-WASI programs. Much like static and dynamic analysis tools, these tools may suffer from a lack of WebAssembly inputs, which could affect the measurement of their effectiveness.

WebAssembly malware: Since the introduction of WebAssembly, the Web has consistently experienced an increase in cryptomalware. This rise primarily stems from the shift of mining algorithms from CPUs to WebAssembly, a transition driven by notable performance benefits [**?**]. Tools such as MineSweeper[**?**], MinerRay[**?**], and MINOS[**?**] employ static analysis with machine learning techniques to detect browser-based cryptomalwares. Conversely, SEISMIC[**?**], RAPID[**?**], and OUTGuard[**?**] leverage dynamic analysis techniques to achieve a similar objective. VirusTotal[14], a tool incorporating over 60 commercial antivirus systems as black-boxes, is capable of detecting cryptomalware in WebAssembly binaries. However, obfuscation studies have exposed their shortcomings, revealing an almost unexplored area for WebAssembly that threatens malware detection accuracy. In concrete, Bahnsali et al. seminal work[**?**] demonstrate that cryptomining algorithm's source code can evade previous techniques through the use of obfuscation techniques.

2.1.7 Open challenges

Despite progress in WebAssembly analysis, numerous challenges remain. WebAssembly, though deterministic and well-typed by design, is susceptible to a variety of security threats. First, most existing WebAssembly research is reactive, focusing on detecting and fixing vulnerabilities already reported. This approach leaves WebAssembly binaries and runtime implementations potentially open to unidentified attacks. Second, side-channel attacks present a significant risk. Genkin et al., for example, illustrated how WebAssembly could be manipulated to extract data via cache timing-side channels [**?**]. Furthermore, research conducted by Maisuradze and Rossow demonstrated the potential for speculative execution attacks on WebAssembly binaries [**?**]. Rokicki et al. disclosed the possibility for port contention side-channel attacks on WebAssembly binaries in browsers [**?**]. Finally, the binaries themselves may be inherently vulnerable. For example, studies by Lehmann et al. and Stiévenart et al. suggested that flaws in C/C++ source code could infiltrate WebAssembly binaries [**? ?**].

[14] `https://www.virustotal.com`

2.2 Software diversification

This dissertation presents toolsets, approaches and methodologies designed to enhance WebAssembly security proactively through Software Diversification. First, Software Diversification could expand the capabilities of the mentioned tools by incorporating diversified program variants, making it more challenging for attackers to exploit any missed vulnerabilities. Generated as proactive security, these diversified variants can simulate a broader set of real-world conditions, thereby making WebAssembly analysis tools more accurate. Second, we noted that current solutions to mitigate side-channel attacks on WebAssembly binaries are either specific to certain attacks or need the modification of runtimes, e.g., Swivel as a cloud-deployed compiler. Software Diversification could mitigate yet-unknown vulnerabilities on WebAssembly binaries by generating diversified variants in a platform-agnostic manner.

Software Diversification has been extensively studied in recent decades. This section explores its current state of the art. Software diversification involves the synthesis, reuse, distribution, and execution of different, functionally equivalent programs. As outlined in Baudry et al.'s survey [**?**], software diversification falls into five usage categories: reusability [**?**], software testing [**?**], performance [**?**], fault tolerance [**?**], and security [**?**]. Our work specifically contributes to the last two categories. This section presents related works, emphasizing how they generate diversification and apply it to WebAssembly.

2.2.1 Generation of Software Variants

Software variants are functionally equivalent versions of an original program, created through software diversification at different stages of the software lifecycle, such as the source code or machine code levels. The diversification can be either natural [**?**], arising spontaneously but requiring significant human effort, or artificial [**?**], which is automated and the focus of our work in the context of WebAssembly. The concept of software variants dates back to Randell's 1975 work [**?**], which introduced the idea of fault-tolerant instruction blocks. Later, artificial software diversification was further developed through rewriting strategies, as proposed by Cohen and similarly by Forrest in the 1990s [**?** **?**]. These strategies consist of rule sets for altering software components to create functionally equivalent but distinct programs. Our work builds on these foundational studies and focuses on artificial diversification techniques, particularly for WebAssembly, drawing insights from significant works by Baudry et al. [**?**] and Jackson et al. [**?**]. In the following, we group the major strategies used to artificially generate software variants and their current state with respect to WebAssembly.

Replacement of Equivalent Instructions: One can replace sections of

programs with semantically equivalent code. This method requires substituting the original code with identical arithmetic expressions or injecting instructions that do not alter the final computation outcome. There are primarily two methods for generating such equivalent code: rewriting rules and enumerative synthesis. In the first method, manual rewriting rules dictate the replacement strategies. A rewriting rule consists of a code segment and its semantically identical substitution. For instance, Cleemput et al. [**?**] and Homescu et al. [**?**] inject NOP instructions to produce statically varied versions. In these studies, the rewriting rule is expressed as `instr => (nop instr)`, implying a `nop` operation followed by the instruction as a valid substitute. In contrast, enumerative synthesis explores all potential programs specific to a language. In this domain, Jacob et al. [**?**] introduced a technique called superdiversification for x86 binaries. Similarly, Tsoupidi et al. [**?**] presented Diversity by Construction, a constraint-based compiler that creates software diversity for the MIPS32 architecture. Their technique employs a constraint solver to generate program variants that are semantically equivalent by design. Compared to other methods, Jacob et al. and Tsoupidi et al.'s work does not need manually written replacement strategies, but their reach is limited by theorem solvers. While their techniques can be implemented in any language, they are not directly applicable to WebAssembly. For instance, while the studies of Cleemput et al. and Homescu et al. are directly applicable to WebAssembly, since WebAssembly typically compiles later, this specific strategy could fall into a *non preserved* category, meaning JIT compilers could eliminate this diversification strategy by simply applying straightforward optimizations. Conversely, the application of enumerative synthesis to WebAssembly has not been explored, and it is actually one of our contributions.

Instruction Reordering: This strategy involves reordering independent instructions or entire program blocks. The location of variable declarations may also change if compilers reorder them in the symbol tables. This prevents static examination and analysis of parameters and alters memory locations. In this area, Bhatkar et al. [**? ?**] proposed the random permutation of variable and routine order for ELF binaries. Such strategies are not implemented for WebAssembly to the best of our knowledge.

Adding, Changing, Removing Jumps and Calls: This strategy generates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [**?**] primarily illustrated this concept by inserting random jumps in programs. Pettis and Hansen [**?**] suggested splitting basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [**?**] de-inlined basic blocks of code as an LLVM pass. In their approach, each de-inlined code transforms into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [**?**] extended their previous approach [**?**],

replacing function calls with indirect pointer calls in C source code, allowing post-binary reordering of function calls. Recently, Romano et al. [**?**] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm functions. As previously discussed in Section 2.1.5, the control flow of WebAssembly is restricted, making them impractical to directly port these strategies to WebAssembly.

Program Memory and Stack Randomization: This strategy alters the layout of programs in the host memory. Additionally, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [**?** **?**] proposes to randomize the base addresses of applications and library memory regions in ELF binaries. Tadesse Aga and Autin [**?**], and Lee et al. [**?**] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [**?**] suggest separating a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [**?**] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization. This makes it very challenging for an attacker to ignore the key to inject executable code. This strategy disrupts the predictability of program execution and mitigates certain exploits such as speculative execution. No work has been found that explicitly applies this strategy to WebAssembly. However, the transformation of WebAssembly binaries inherently randomizes the memory layout. Consequently, memory access is randomized as these binaries are further JITed to machine code in the majority of cases.

ISA Randomization and Simulation This strategy involves using a key to cypher the original program binary into another encoded binary. Once encoded, the program can only be decoded at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [**?**], and Barrantes et al. [**?**] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [**?**] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [**?**] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [**?**] can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client. Yet, both strategies aim to tackle program analysis from potential attackers. Moreover, this strategy faces a performance penalty, specially for WebAssembly, due to the decoding process as shown in WASMixer evaluation [**?**].

Equivalence checking between program variants is a vital component for any program transformation task, ranging from checking compiler optimizations [**?**] to the artificial synthesis of programs discussed in this chapter. It proves that two pieces of code or programs are functionally equivalent [**?**]. Cohen [**?**] simplifies the checking process with the following property: two programs are equivalent if, given identical inputs, they produce identical outputs. We adopt this definition of *functional equivalence* throughout this dissertation. In Software Diversification, equivalence checking seeks to preserve the original functionality of programs while varying observable behaviors. Two programs, for instance, can differ statically and still compute the same result. We outline two methods to check variant equivalence: by construction and automated equivalence checking. In Chapter 3, we discuss the primary advantages and limitations in practice for both approaches within the scope of our contributions.

Equivalence checking by construction: The equivalence property is often guaranteed by construction. Cleemput et al. [**?**] and Homescu et al. [**?**], for example, design their transformation strategies to generate semantically equivalent program variants. However, developer errors can occur in this process, necessitating further validation. The test suite of the original program can serve as a check for the variant. If the program variant passes the test suite [**?**], it can be considered equivalent to the original. However, this technique is limited by the need for a preexisting test suite and does not give guarantees. Should the test suite not exist, an alternative technique is required for equivalence checking. An alternative method for checking program equivalence involves the use of fuzzers [**?**]. Fuzzers randomly generate inputs that yield different observable behaviors. If two inputs produce a different output in the variant, the variant and the original program are not equivalent. The primary limitation for fuzzers is that the process is notably time-consuming and necessitates manual introduction of oracles.

Automated equivalence checking: In the absence of a test suite or a technique that inherently implements the equivalence property, the works mentioned earlier use theorem solvers (SMT solvers) [**?**] to prove equivalence. The central idea for SMT solvers is to convert the two code variants into mathematical formulas. The SMT solver then checks for counter-examples. When it finds a counter-example, there is an input for which the two mathematical formulas yield different outputs. The primary limitation of this technique is that not all algorithms can be translated into a mathematical formula, such as loops. Nevertheless, this technique is frequently used for checking no-branching-programs like basic block and peephole replacements [**?**].

2.2.2 Variants deployment

Program variants, once generated and verified, may be utilized in two primary scenarios: Randomization or Multivariant Execution (MVE) [**?**]. Additionally,

these variants serve both defensive and offensive purposes [**?**].

Randomization: In the context of our work, the term *Randomization* denotes a program's ability to present different variants to different clients. In this setup, a program, chosen from a collection of variants (referred to as the program's variant pool), is assigned to a random client during each deployment. Jackson et al. [**?**] define the variant pool in Randomization as herd immunity, as vulnerable binaries can only affect a segment of the client community. El-Khalil and colleagues [**?**] suggest employing a custom compiler to generate varying binaries from the compilation process. They adapt a version of GCC 4.1 to partition a conventional stack into several component parts, termed multistacks. Similarly, Singhal and colleagues, propose Cornucopia [**?**]. Cornucopia generates multiple variants of a program by using different compiler flag combinations. Aga and colleagues [**?**], contributing to this discussion, propose the generation of program variants through the randomization of its data layout in memory. This method allows each variant to operate on the same data in memory but at different memory offsets. Randomization can also be applied to virtual machines and operating systems. On this note, Kc et al. [**?**] establish a unique mapping between artificial CPU instructions and actual ones, enabling the assignment of various variants to specific target clients. In a similar vein, Xu et al. [**?**] recompile the Linux Kernel to minimize the exposure time of persistent memory objects, thereby increasing the frequency of address randomization.

Multivariant Execution (MVE): Multiple program variants are composed into a single binary, known as a multivariant binary [**?**]. Each multivariant binary is randomly deployed to a client. Then, the multivariant binary executes its embedded program variants at runtime. These embedded variants can either execute in parallel to check for inconsistencies, or as a single program to randomize execution paths [**?**]. Bruschi and colleagues extend the concept of executing two variants in parallel, introducing non-overlapping and randomized memory layouts [**?**]. At the same time, Salamat et al. modifies a standard library to generate 32-bit Intel variants. These variants have a stack that grows in the opposite direction, allowing for the detection of memory inconsistencies [**?**]. Davi and colleagues propose Isomeron, an approach for execution-path randomization [**?**]. Isomeron operates by simultaneously loading the original program and a variant. It then uses a coin flip to determine which copy of the program to execute next at the function call level. Previous works have highlighted the benefits of limiting execution to only two variants in a multivariant environment. Agosta and colleagues, as well as Crane and colleagues, used more than two generated programs in the multivariant composition, thereby randomizing software control flow at runtime [**?** **?**]. Both strategies have proven effective in enhancing security by addressing known vulnerabilities, such as Just-In-Time Return-Oriented Programming (JIT-ROP) attacks [**?**] and power side-channel attacks [**?**]. Lastly, only Voulimeneas et al. [**?**] have recently

proposed a multivariant execution system that enhances security by parallelizing the execution of variants across different machines.
TODO Redo this part

Defensive Diversification: Lundquist and colleagues [**?**] separate the usages of Software diversification into two categories: Defensive Software Diversification and Offensive Software Diversification. Defensive Software Diversification is the traditional application of previously discussed techniques aimed at enhancing the security and reliability of software systems. The core idea is to make it more difficult for attackers to predict the system's behavior and exploit program vulnerabilities. By doing so, even if an attacker manages to compromise one variant, the others may remain secure, thereby limiting the potential impact of the attack. Defensive diversification is often complementary with other security measures, such as encryption and intrusion detection systems, to create a multi-layered defense strategy [**?**].

Offensive Diversification: Offensive Software Diversification uses the principles of software diversification. Yet, in the offensive context, diversification techniques may be applied to malware or other malicious code to evade detection by security software [**?**]. For example, in the context of Wasm, the seminal work of Romano et al. [**?**] proposed to intermix JavaScript and Wasm code to obfuscate JavaScript malware and make it more difficult to analyze for malware detectors. The obfuscated version of the code is functionally equivalent to the original, being, in practice, Offensive Software Diversification. Notice that, this method also measures the resilience and accuracy of security systems. However, obfuscation studies have exposed their shortcomings, revealing an almost unexplored area for WebAssembly that threatens malware detection accuracy. In concrete, Bahnsali et al. seminal work[**?**] demonstrate that cryptomining algorithm's source code can evade previous techniques through the use of obfuscation techniques.

2.2.3 Open challenges

As outlined in Section 2.1.7, our primary motivation for the contributions of this thesis is the open issues within the WebAssembly ecosystem. We see potential in employing Software Diversification to address them. Based on our previous discussion, we highlight several open challenges in the realm of Software Diversification for WebAssembly. First, Wasm being an emerging technology, is still in the process of implementing defensive measures [**?**]. The process of officially adopting a new defensive measure is inherently slow, making software diversification a potentially valuable preemptive strategy. Second, despite the abundance of related work on software diversity, its exploration in the context of Wasm remains limited. Third, both defensive and offensive software diversification have been largely unexplored. Notably, the works on

malware detection discussed in Section 2.1.6 suggest that offensive diversification could be useful in measuring the resilience and accuracy of security systems for WebAssembly.

Conclusions

In this chapter, we presented an overview of the Wasm language. This included its binary format, runtime execution concepts, and security issues. Related work was also discussed. The goal of this chapter is to establish a foundation for studying automatic diversification in Wasm. We emphasized the fact that Wasm has not been extensively researched in the field of Artificial Software Diversification. Existing implementations for Software Diversification cannot be directly applied to Wasm. Current security limitations and the absence of software diversity approaches for Wasm inspire our work. In Chapter 3, we elaborate on the technical details that guide our contributions.